1-1-2005

Secure group communication protocol and implementation for JetMeeting, an application based on P2P

Yan Li
Iowa State University

Secure group communication protocol and implementation for JetMeeting, an application based on P2P

by

Yan Li

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Douglas Jacobson, Major Professor
Peter Boysen
James Davis

Iowa State University

Ames, Iowa

2005

Graduate College
Iowa State University

This is to certify that the master's thesis of

Yan Li

has met the thesis requirements of Iowa State University

TABLE OF CONTENTS

ABSTRACT

Supporting secure applications in a distributed environment faces several challenges. Scalable security in the context of multicasting is especially hard when privacy and authenticity is to be assured to highly dynamic groups where the application allows participants to join at any time.

The proposals for multicasting security solutions that have been published so far are complex, often are inefficient. In this thesis, we propose a security protocol for achieving scalable security within JetMeeting, an application based on a P2P communication. Our solutions assure that newly joining group members are not able to use the group secure services without the group key.

For versatility, our secure protocol supports a scheme for key management, which is devised for Group Key Manager election, Group Key generation, periodic rekey and encrypted transmission. Operations have low complexity for joins with low overhead, thus granting scalability even for large groups.

In this thesis we present our implementation based on the existing cryptographic methods. The benefits of this implementation are that it provides the dynamic and decentralized group key management, minimizes the transmissions required to distribute the initialized group key and to rekey the multicast group, as well as multiple simultaneously get the same transmission with one group key. The system was evaluated by experimental tests, and the results provide evidence of its efficiency and scalability.

1. INTRODUCTION

The technology that enables the distributed computing is called peer-to-peer computing or peer-to-peer networking; most people simply refer to it as P2P. Aiming to ease the problems associated with making a variety of items available to multiple users over the Internet, P2P describes an environment where computers connect to each other in a distributed environment that does not use a centralized control point to route or connect data traffic.

Many factors today make P2P practical for a wide number of applications. These factors include the explosion of connected devices, the rapid increase of affordable bandwidth acceleration of computing power, lager storage capacities. P2P is applied to a wide range of technologies that greatly increase the utilization of information, bandwidth, and computing resources in the Internet.

In practice, P2P technologies deployed today adopt a network-based computing style that neither exclude nor inherently depends on centralized control points. In general, P2P is more a style of computing that makes the network interactions more symmetrical.

P2P gives peers control to use and access their data as they see fit. P2P applications are flexible and tolerance of errors. They replicate data as needed and broadcast data to multicast computers.

1.1 JXTA

JXTA provides a P2P platform technology and a community of resources necessary to develop new P2P applications. Unlike first-generation peer-to-peer networks, which are tired to a specific application, JXTA is an open network computing platform designed for P2P computing and mostly developed by Sun Microsystems. JXTA's goal is to develop basic building blocks and services to enable innovative applications for peer groups [1-3]. JXTA technology is a network programming and computing platform that is designed to solve a number of problems in modern distributed computing, especially in the area broadly referred

to as peer-to-peer networking [4]. JXTA framework provides a set of protocols and a series of services that let peers to find each other, form groups and directly exchange messages [1].

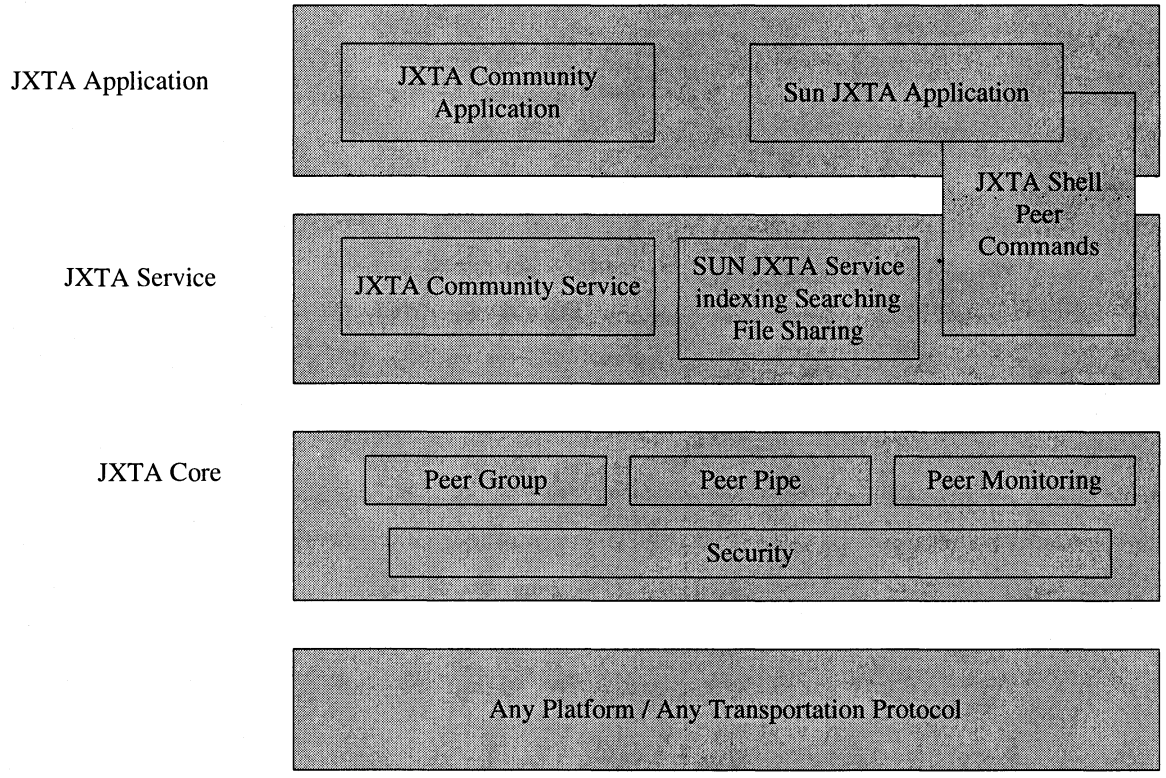

Figure 1. JXTA Architecture

JXTA software architecture is divided into three layers, as shown in figure 1.

- **Platform Layer**, also known as the JXTA core, encapsulates minimal and essential primitives that are common to P2P networking. It includes building blocks to enable discovery, transport with firewall handling, the creation of peers and peer groups, and associated security primitives.
- **Service Layer** includes network services that may not be absolutely necessary for a P2P network to operate, but are common or desirable in the P2P environment. Examples of network services include searching and indexing, directory, storage systems, file sharing, distributed file systems, and Public Key Infrastructure (PKI) services.
- **Application Layer** includes implementation of integrated applications, such as P2P instant messaging, document and resource sharing.

JXTA network is an ad hoc, multi-hop, and adaptive network composed of connected peers as shown in figure 2. JXTA peers advertise their services in XML documents called advertisements. Advertisements enable other peers on the network to learn how to connect to, and interact with, a peer's services. JXTA peers use pipes to send messages to one another. A peer can send and receive messages, and will cache advertisements. A rendezvous peer is like any other peer, however, it also forward discovery requests to help other peers discover resources. A relay peer maintains information about the routes to other peers and routes messages to peers. A peer first looks in its local cache for more route information. If it isn't found, the peer sends queries to relay peers asking for route information. Relay peers also forward messages on the behalf of peers that cannot directly address another peer (e.g. NAT environments), bridging different physical and/or logical networks.

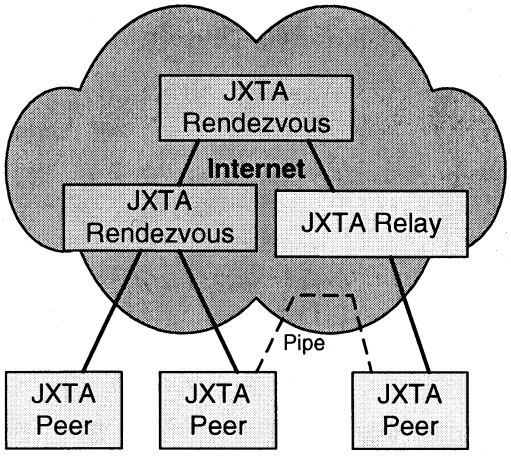

Figure 2. JXTA Network

JXTA supports protocols other than HTTP, such as TCP and IP broadcasting, so it is able to choose the most efficient protocols for each situation. Using the JXTA protocols, peers can cooperate to form self-organized and self configured peer groups independently of their positions in the network (edges, firewalls), and without the need of a centralized management infrastructure.

1.2 JetMeeting

JetMeeting, a project undertaken by Academic Information Technologies at Iowa State University, is a web-based application that provides P2P synchronous services to uPortal

groups. *JetMeeting* can be used to hold meetings with other students and/or the instructor and permits the sharing of text, audio, graphics and applications. *JetMeeting* focuses on multicast and dynamic group communication in which a group comprises peers with a specific common interest within the network.

The usage paradigm of *JetMeeting* tool is very similar to that of entering a group meeting room. The peer joins a group and uses one or more of the collaborative tools including chat, whiteboard, audio or application sharing in the process of the meeting. Each peer joins a unique group for the meeting and from then on all communication happen over the JXTA platform. *JetMeeting* is a Java application that is in the JXTA's application layer. *JetMeeting* architecture is illustrated in Figure 3.

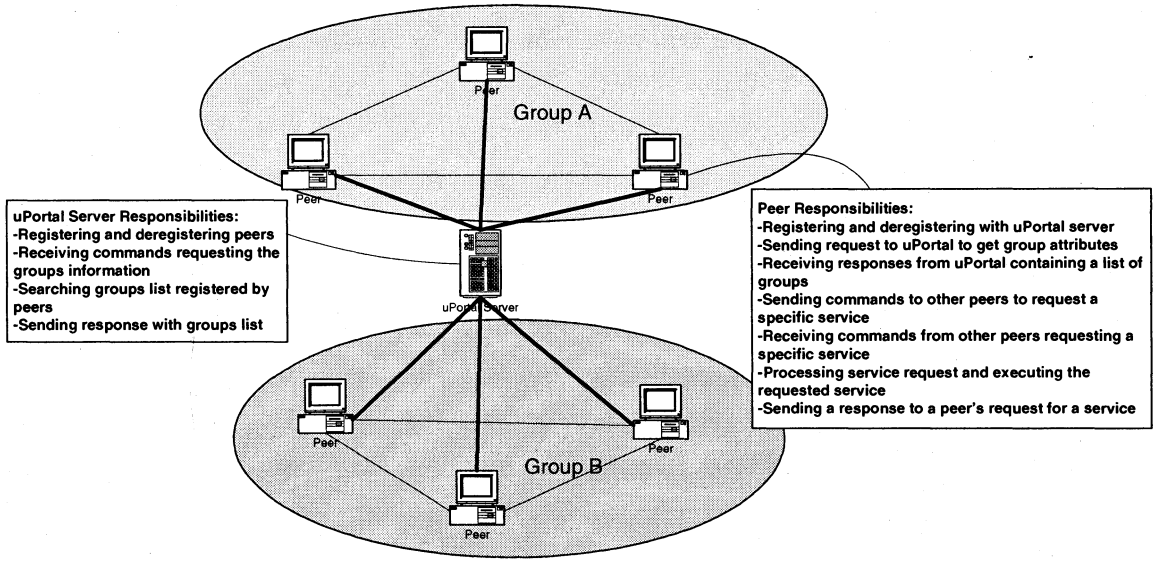

Figure 3. JetMeeting Architecture

JetMeeting is not a pure P2P system and uses a central server, uPortal, to verify the identity of the login peers when new peers connect to *JetMeeting*. All other communication between peers is conducted in a P2P fashion, with messages flowing directly from one peer's machine to another's without uPortal intermediary. uPortal server is responsible for interacting with the back-end database that maintains all the data pertaining to the system. The peers log in to the uPortal server through a web interface and access a set of tools that enable group

management. Once the group management is enabled, *JetMeeting* tool provides the fully decentralized collaborative environment to the peers, which communicate with each other directly within one group, rather than through uPortal. *JetMeeting* has been a scalable and efficient system for messaging collaborating.

Chat is one of major tools of *JetMeeting*. The most popular commercial chat solutions use a centralized implementation. In short, all interactions go through one or more central servers that provide an accurate directory of connected members and route all messages. The JXTA technology is ideally suited for chat implementations. The fully-decentralized solution was the simplest because JXTA platform takes care of all the underlying issues to discover and communicate with other peers. JXTA handles the discovery of other nodes and the secure routing and exchange of user and inter-application messages among them.

The uPortal has basic security solutions for peers' and groups' management. Once the peer logs in uPortal server, the system checks the database to make sure which groups this peer belongs to, then assigns this peer with the group IDs and group attribution, such as secure group chat or not secure group chat. The above information ensures that the peers can only join the limited groups and work in a trusting environment but the communication between peers is not guaranteed to be privacy and integrity.

1.3 Secure Group Chat Scenario

There is a scenario that stresses the group key management infrastructure. This scenario is secure group chat in *JetMeeting,* which deals with the peer's need to communicate with the other peers with encrypted multicast messages. Secure group chat is more demanding from a key management perspective than other *JetMeeting's* tools for several reasons. First, the group members of secure group chat need to simultaneously keep the same group key. The group should have a manager to generate and periodically update the group key. The group key needs to be distributed securely and automatically, but not manually. This scenario involves that the group members require to dynamically join or to leave from the group within one second. The secure group chat scenario has a fairly thorough set of security

requirement, covering access control, member to member authentication, data confidentiality, and data integrity. Finally, the notion of availability to this scenario, which means that the peer, who provides the group key management service, must be up.

1.4 Security Issues

Security of most existing P2P platforms is not as critical as for enterprises which weigh adopting P2P for collaboration across their organizations as well as in the Internet. As a consequence they do not address security issues to the extent that is required by the enterprises [5]. Current solutions often require human intervention (manual keying is common), or restrict the dynamics provided by multicasting and required by many applications. Multi-party communication applications based on dynamical and decentralized P2P networks introduce a new bunch of security issues.

The secure multicast requirement is the necessary for multiple users who share the same security attributes and communication requirements to securely communicate with every other member of the multicast group using a group key. The largest benefit of the multicast communication is that multiple receivers simultaneously get the same transmission. Thus the problem is enabling each user to determine/obtain the same group key without permitting unauthorized parties to do likewise (initializing the multicast group) securely rekeying the users of the multicast group when necessary.

This thesis shows that multicast communications have dynamically changing requirements, which will make it very challenging requirements from a key management perspective to address. The purpose of this thesis is to develop key management schemes, which guarantee that at each instance in time only actual group members will be possession of the cryptographic key needed to participate. This thesis aims at applying the concepts of group key management into *JetMeeting* in order to provide an efficient and a scalable security infrastructure. The employment of the protocol will be described in the following sections.

The thesis starts by explaining group key management issues and the model of threat we are concerned about and the various sorts of security services we can provide. Next we provide an overview of cryptographic algorithms how to put them together to implement these security services. Then we evaluate the system efficiency, fault tolerance and scalability. This thesis draws conclusions and explores a summary some possible enhancements to this system.

2. GROUP KEY MANAGEMENT ISSUES

There are many factors that must be taken into account when developing the desired key management architecture. Important issues, particular to multicast groups include:

- What are the security requirements of the group members? Most likely there will be some group key manager, or managers.
- How does the formation of the multicast group occur? Will the group key manager initiate the group member joining process, or will the group members initiate when they join the multicast group?
- One must minimize the number of bits required for multicast group key distribution. This greatly impacts bandwidth-limited equipments.

All of these issues need to be taken into account, along with a threat model. The threat model describes what resources we expect the attacker to have available and what attacks the attacker can be expected to mount [6]. Nearly every security system is vulnerable to some threat or another.

JetMeeting assumes a threat model and wants to protect the communications. The group member is threatened by outside attacks such as sniffing. It assumes attacker snoop communication and retrieve confidential information. Some may even impersonate businesses and spoof and defraud others. *JetMeeting* security is designed to ward off attacks ranging from traffic analysis to eavesdropping to message tampering.

In addition to the threat model, we made two deployment assumptions. First, no central security administration is available. The security system should operate in a decentralized environment. Second, the security infrastructure should scale up to hundreds of members and the members dynamically to join and leave the group.

In our system, we adopt a security protocol that relies on existing and trusted technologies. Such technologies are made practical as a consequence of two choices recently made: the adoption of X.509 Digital Certificate and the existing encryption/decryption for the secure

transport of information; the exploitation of the end-to-end transport independence of JXTA protocol. The advantages of this approach are clear. Because no new invention is required, plans for applications development and deployment can proceed with minimal delay.

The threat model together with the existing trusted technologies and assumptions leads us to the cryptography-based security protocol (in particular, PKI) with unrestricted certificate issuance and end-to-end message encryption and authentication. In this thesis, we concentrate on a secure, efficient, scalable and practical protocol.

The security protocol provides mechanisms to disseminate group policy and generate group key. To protect the privacy of the data not to be tampered by the attackers, encryption is the most direct way to address this issue. Because asymmetric encryption consumes more processing time and memory for calculations than symmetric key encryption, a combination of symmetric and asymmetric is used in our security protocol. Public key encryption is used to securely exchange a set of symmetric key materials. After this is done, symmetric encryption can be used to encrypt the service data with the symmetric key. This combination of public key encryption and symmetric encryption provides fast message encryption with the benefits of certificate-based key management.

The major issue of this security protocol is the key management in dynamic groups. The key for multicast services is called Group Key (GK). Group key management, especially in a group for multicasting is the corner stone for all other security services [7]. We will set up a key management protocol to manage the group keys with a dynamic distributed approach. This approach works well for honest groups and not very large groups.

For multicast security, several key management schemes have been proposed, e.g. the Group Key Management Protocol (GKMP)[8][9], and the Internet Key Exchange (IKE) [10]. They are complex and inefficient in dealing this issue. Our group key management approach efficiently and securely distributes a Group Key to members as they join.

3. GROUP KEY MANAGEMENT PROTOCOL

This section presents a group key management protocol in this security system. It is run between a group member and a Group Key Manager (GKM), which establishes security associations among authorized group members.

The protocol described below, delegates key generation and distribution functions to the members themselves rather than relying on a third party for these functions. As prelude to actually distributing key, a few things must be assumed: there exists a reliable "group key manager" responsible for creating and distributing keys to group members with certification.

Figure 4 shows this group key management's architecture, which is consisted of 4 functions: GKM Election, GK Generation, GK Distribution, and Rekey.

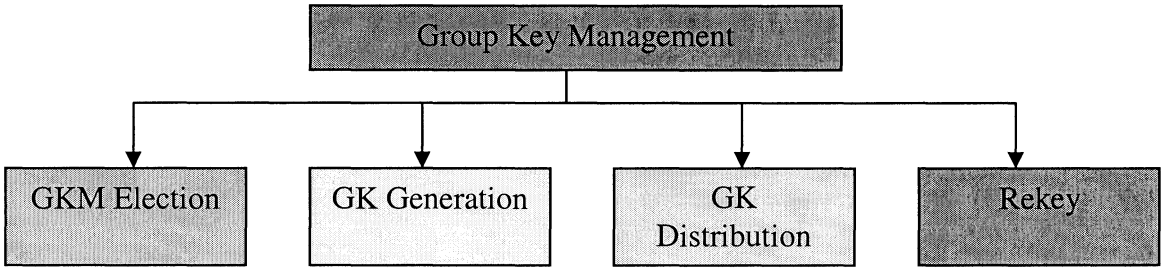

Figure 4. Group Key Management Architecture

For secure group communications to take place, all members must obtain the same key. This may be achieved by both using deterministic key generation techniques (Using a secret, shared seed) and by making one member of the group responsible creation of the key [8]. The use of a deterministic key generator presents security problems, particularly regarding lots of the seed. The assignment of a member to the role of key manager also presents drawbacks, but these relate to determining which one should be the manager and the need for each group member to contact it. Below describes the Group Key Manager election.

3.1 Group Key Manager

The Group Key Manager is a group member with authority to perform critical actions. GKM is made responsible for initial group key and periodic generation and distribution of new

rekey message. All group members dynamically have the capability to be a GKM and could assume this duty upon assignment. The Group Key Manager should be active for all time and only one Group Key Manager should be active for the entire group. The GKM helps the cryptographic group reach and maintain key synchronization. A group must operate on the same symmetric cryptographic key. If part of the group loses or inappropriately changes the key, it will not be able to send or receive data to another member on the correct key. Therefore, it is important that those operations that create or change the key are unambiguous and controlled. The GKM interacts with other management functions in the network to provide the group key management with group membership lists and group relevant commands. The GKM deals strictly with cryptographic.

3.2 Group Key Generation

The Group Key Manager creates a group key for use in the group. The creation of a cryptographic key requires that the key be sufficiently random. Cryptographic randomness doesn't mean just statistical randomness, although that's part of it. For a sequence to be cryptographically secure pseudo-random, it must be unpredictable what the next random bit will be, given complete knowledge of the algorithm or hardware generating the sequence and all of the previous bits in the stream [11]. The key is generally the seed used to set the initial state of the generator.

Like any cryptographic algorithm, cryptographically secure pseudo-random-sequence generators are subject to attack. Making generators resistant to attack is what cryptography is all about.

To generate random numbers, we find a whole lot of seemingly random events and distill randomness from them. This randomness then is stored in a pool that applications can draw on as needed. SHA-1 hash function is ready-made for the job; it is fast, so we can shovel quite a bit through them without worrying too much about performance or the actual randomness of each observation. Hash function takes an arbitrary amount of input and produces an output mixing all the input bits. The use of multiple random inputs with a strong

hash function can overcome weakness in any particular input. Using an "encryption" algorithm with a random key and seed value to encrypt and feedback the output encrypted value into the value to be encrypted for the next iteration is the mechanism to generate a secure key in our security protocol.

3.3 Group Key Distribution

After the group key is generated, the GKM calls a listener to listen other group members' key request message. Before the group key is distributed, the GKM must verify the identity and permissions of each member prior to the key being distributed. Likewise, the group member must verify GKM's identity to perform this action and permissions.

The key being distributed is the same secure level as the data that it will encrypt. Hence, we must encrypt the key during distribution. To implement a successful security key distribution, here needs to be a strong notion of identity to identify the group members. This requirement suggests the use of a full Public Key Infrastructure (PKI) with the use of Certification Agencies (CA) [12].

PKI is a good fit for JXTA because X509 certificates are robust enough to allow peer groups, peers services to be uniquely identified, authenticated and scoped to specific roles. The private key is used to create something called a digital signature and public key encryption bears to secret-key encryption; sender use its private key to sign a message and the receiver uses sender's public key to verify the sender's signature. The digital signature has one important property: nonrepudiation. The recipient can prove that the sender signed the message and the sender cannot deny it.

Digital certificate comes from an X509 CA willing to vouch for an identity and public-key pair. CA policies vary, depending on the strength of authentication required by users and the CA's specific functional requirements. The notion of a centralized CA is not appropriate to the world of peer-to-peer application and is at times contradictory. Large numbers of peers could want to conduct a secure transaction without involving a centralized infrastructure.

In our system, we let peers become their own certificate, authorities; generate their own root certificate that verifies that they are associated with a public key, which is presented with a pair of values: modulus and public exponent. The length of these pair keys is 2048 bits. The peer holds the private key for the certificate; the subject of the certificate describes the peer, its identifier and its creator. A fingerprint hash of the public key embedded in the certificate is used as the source of ID providing a tight binding between the identifier and the authenticator. The public key data associated with the certificate can be used to sign and encrypt message traffic. The expiry date of the certificate is on the order of the lifetime of the peer. When a PKI is implemented, the private key must be kept in a secure data store accessible only to the JXTA peer that is authorized to access it.

```
Version: 3
IssuerDN: O=www.jxta.org, L=SF, C=US, CN=jnushareuser-CA,
        OU=99CEBA4D8294A0493C8A
Start Date: Sat May 22 15:36:57 EDT 2004
Final Date: Thu May 22 15:36:57 EDT 2014
SubjectDN: O=www.jxta.org, L=SF, C=US, CN=jnushareuser-CA,
OU=99CEBA4D8294A0493C8A
Public Key: RSA Public Key
Modulus:
8b48e5586350a4626eb4e2b4fd99ae63c85bddf30
0b6340e49a192ce0bf595d8c74a6182e14ade29ba
f63bbd9a92f8d5cc5729f7eb1e21383f20454b412
db306c1b3b50cd7cb8745bbdc451fadcadb0cccec7
d8d366a4accb3424cc2a86a6cad2ebd24fc3bb26c7
7d6debecd7cae5f872a4cacbefb280ea68700adb6b
404e87d
public exponent: 11
Signature Algorithm: SHA1WithRSAEncryption
Signature: 8594de482796d532ad42bcf61131dcb686c2e2c2
        35db7700df63e74addb8afd9d1196b5ed0e3ecd5
        e9abb1864c63232c4645a8c598e6982f90158961
        4c7ecce3d5ac00de4f02395b043fba6983a349dc
        3429be3f80cb0270d604753f39cc29d0fe6d552c
        7550175447c68ef209f057a12eedadc53bb3948d
        c268fd22ddbac003
```

Figure 5. X.509 Certificate Format

X.509 is the most widely used standard for digital certificates. In our system, the certificate is represented as figure 5. It shows a view of certificate generated by peer's own.
JXTA allows to establish a secure communication which provides security by means of cryptographic materials: X.509 certificates and RSA keys. This material is generated when JXTA is launched and then stored in the JXTA's Personal Security Environment (pse) directory (pse/). This certificate is included in the peer advertisement.

A secure communication should provide peer authentication, encryption and message integrity [12]. It is divided into two phase, the handshake and data transfer phase. The handshake phase authenticates the GKM and the group member, exchanges certificates, which are used to protect the data to be transmitted. The handshake must be completed before any application data can be transmitted. Once this has been done, the group key is transmitted as a series of protected records.

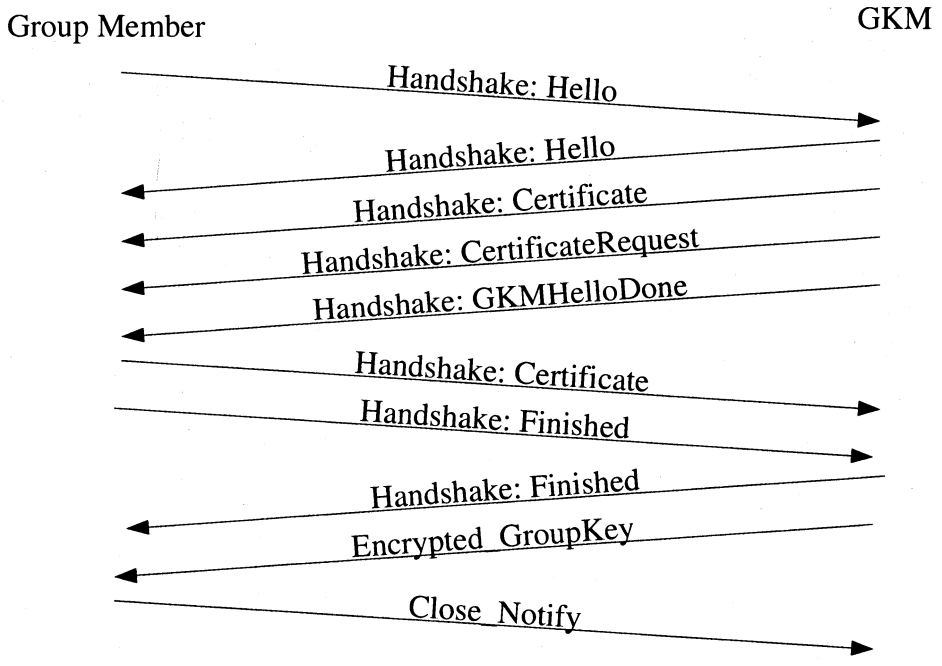

Figure 6. Time sequence of Secure Communication

The idea here is that the peer uses SHA-1 to hash the certificate and its private key to encrypt the content of the certificate, which is stored as signature in the certificate, thus proving that the peer has possession of the private key corresponding to the certificate and the certificate content is not revised during the transmission. The group member authenticate is initiated by GKM sending a *CertificateRequestMsg* message to the group member. The group member responds by sending a *CertificateMsg* message, which is a string signed with Sha-1 hash and RSA algorithms. Figure 6 shows the timeline for this process.

The purpose of the handshake is that GKM and group member authenticate each other. The overall process works like this:

- The group member and the GKM exchange their certificates, which include the public keys.
- Verify the certificates and extract the public keys.
- GKM encrypts the group key with an asymmetric encryption method.

The above process accomplishes two goals, first, is to authenticate the GKM and the group member as well as provide the certificates. Steps 1 and 2 accomplish the first goal. Steps 2 and 3 accomplish the second goal, establishing secure communications. In step 2 group member provides the GKM with its certificate, which allows group member to transmit a secret to GKM. After step 2, both of the group member and the GKM keep each other's public key.

Note that step 3 is the key step in the whole process. What is going on is very simple: the GKM uses the group member's public key (extracted from the certificate) to encrypt the group key for data integrity, and GKM uses its private key to re-encrypt the encrypted group key for authentication. The rest of the handshake is mainly devoted to ensuring that this exchange can happen safely. The entire process serves to protect the delivering group key procedure itself from tampering. Imagine an attacker who wished to pretend to be the group member to steal the group key. The attacker could not get the group key because it does not have the group member's private key to decrypt the message.

4. GROUP KEY MANAGEMENT PROTOCOL IMPLEMENTATION

The Group Key Management protocol is used to establish secure communications among group members within a group. In this section, we will look into the details of the implementation of group key management and of how to establish secure communications.

The implementation of secure communication within a group has been developed using Java 1.4 and JXTA 2.1 to be platform-independent. The purpose of this security protocol and implementation is to make the *JetMeeting* application based on P2P to have a secure group communication and manage the group key within a dynamic group. Initial experience with the protocol has enabled to improve the security of *JetMeeting*.

4.1 Create a peer group

Before a peer becomes a member within a group, it needs to access uPortal to be authorized. UPortal verifies the peer's authentication, gets the information of groups, which the peer belongs to. Below shows the response information from uPortal:

Group Name, Group ID, Specification;

Group Name, Group ID, Specification;

.....

How peers are authenticated by uPortal is not this thesis's business.

To create a new peer group, the first thing to do is to construct a peer group advertisement. A peer group advertisement describes the peer group and contains the following fields.

- *Peer group ID*: the URN uniquely identifies the peer group.
- *Specification*: This is used to retrieve the module implementation advertisement of the group, which contains a list of all the services available within the group. *JetMeeting* can provide secure group communication and non-secure group communication. We use the module spec ID to identify group communication types.

• *Description*: the description of the group.

Based on the information coming from uPortal, the content of the group advertisement is generated. From the peer group advertisement, the group can be instantiated.

4.2 Join a Group

Once a group member is authenticated, it needs to be able to use services within the group. Joining a group is done in four steps using the secure group membership service. The secure group membership services consist of GKM election, Group Key generation and Group Key distribution. Within the group, all the membership services match the JXTA two steps scheme for membership appliance. Figure 7 shows the joining procedure.

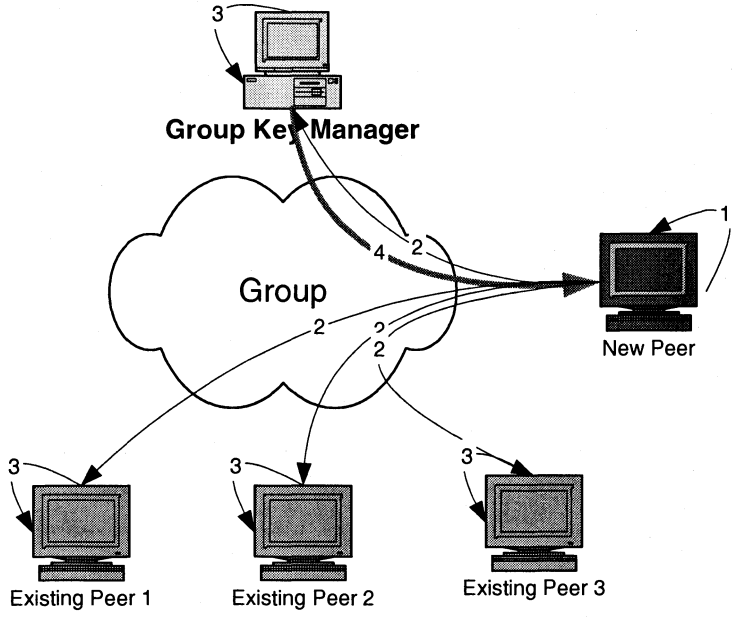

Figure 5. New Peer Joins a Group

Step 1: The newly joining peer calls *apply()* method to generate a *WhoIsGKMMsg* message. See figure 6.

Source ID: Newly Joining Peer
Message Type: WhoIsGKM

Figure 6. *WhoIsGKMMsg* Format

Step 2: The new peer broadcasts *WhoIsGKMMsg* over the group. Broadcast messaging is very economical because there is only one message sent. Broadcasting is very useful there are many members that could answers a query, but you don't know which one has the answer.

Step 3: When a peer receives the *WhoIsGKMMsg*, it checks its Boolean value *GKM*. If this value is True, it means that this peer is the Group Key Manager, and then it generates a *GKMConfirmedMsg* (As shown in figure 7). If the peer's Boolean value *GKM* equals to false, it means that this peer is not the GKM, and then it just discards the *WhoIsGKMMsg*.

Step 4:GKM sends *GKMConfirmedMsg* back to the newly joining peer.

Source ID: GKM
Destination ID: Newly Joining Peer
Message Type: GKM Confirmed

Figure 7. *GKMConfirmedMsg* Format

After the new peer receives the *GKMConfirmedMsg* from the GKM, it starts the Group Key requirement procedure. If it does not receive any response after it sends out *WhoIsGKMMsg*, this means that the current group does not contain a GKM. The newly joining peer is the first peer in the group. So this peer calls the GKM election function, then it is automatically elected as the Group Key Manager and calls the listener to wait for other peer's *WhoIsGKMMsg*.

4.3 Elect a Group Key Manager

After the group has been instantiated, a peer needs to become a Group Key Manager within this group. Electing the Group Key Manager's policy is: the peer who is the first one to join the group will be automatically elected as the Group Key Manager within the group. If the Group Key Manager leaves the group, it will pass the Group Key Manager's responsibility to the peer who is the last one to join the group.

The scenario presented below is a brief summary of the Group Key Manager election procedure.

Newly joining peer Alice sends out a *WhoIsGKMMsg* on the network, if no one reply the inquiry, Alice configures itself as the Group Key Manager. The GKM configuration looks like this:

Boolean GKM = True;

MemberList: records the IDs of the joined group members

After the Group Key Manager is configured, it calls the *listener()* to listen the *WhoIsGKMMsg* and calls *GenerateKey()* function to generate the group key, which is used for secure communications within the group. If GKM receives the *WhoIsGKMMsg*, it calls *GroupKeyDistribution()* function to distribute the group key with a secure way.

When the current Group key Manager, Alice, plans to leave the group, before leaving, it selects peer Bob who is the last one to join the group to be the new Group Key Manager, and sends a *GKMLeavingMsg* (Figure 8) to Bob.

Source ID: GKM
Destination ID: Bob
Message Type: GKM Leaving
Content: Member List

Figure 8. *GKMLeavingMsg* Format

When Bob receives the GKMLeavingMsg, it changes its Boolean value *GKM* to be True, and calls the listener to wait for the *WhoIsGKMMsg* and sends back a confirmation to inform the former GKM that Bob has been the new GKM. When Alice receives the confirmation, it changes its GKM value to False and then leaves the group.

Figure 9 describes the procedure about how to re-elect a new Group Key Manager.

When a new peer joins in the group and broadcasts the GKMRequestMsg, Bob will response the GKMConfirmedMsg as the new Group Key Manager.

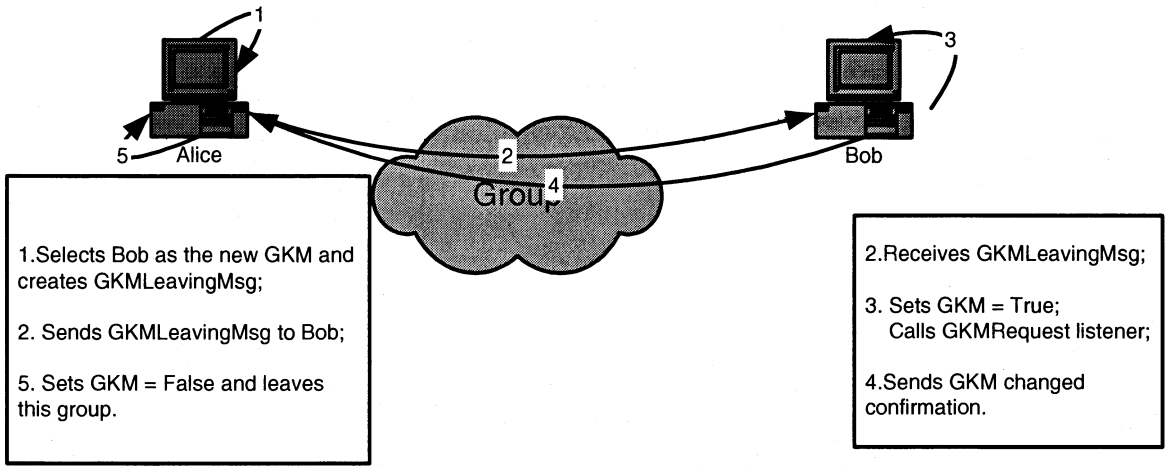

Figure 9. Re-elect a New Group Key Manager

4.4 Generate a Group Key

There's already a random-number generator built into Java, a mere function call away. This random-number generator is almost definitely not secure enough for cryptography, and probably not even very random. Our secure group chat is extremely sensitive to the properties of random-number generator, who generates the Group Key.

This system demands much more of a pseudo-random-sequence generator than do most other applications. Cryptographic randomness doesn't mean just statistical randomness, although that's part of it. For a sequence to be cryptographically secure pseudo-random, it must have this property:

> *It is unpredictable. It must be computationally infeasible to predict what the next random bit will be, given complete knowledge of the algorithm or hardware generating the sequence and all of the previous bits in the stream. [14].*

The output of a generator satisfying this property will be good enough for the key generation, and the system, which requires a truly random sequence generator. The primary point is to generate a sequence of bits that the attacker is unlikely to guess.

To achieve a cryptographically secure pseudo-random sequence, the best way is to find a whole lot of seemingly random events and distill randomness from them. This randomness can then be stored in a pool that applications can draw on as needed. Hash function, SHA-1, is ready-made for the job; so we can shovel quite a bit through SHA-1 without worrying too much about performance or the actual randomness of each observation.

To generate secure pseudo random data, a much better initial seed must be used. Seed is our random number and generates our next random number. Here we use the recommendation of rfc2412 for a choice of a seed. These are the first 160 bits of the binary expansion of pseudo integer [13]:

```
private static byte[] seed = {
(byte) 0xC9, (byte) 0x0F, (byte) 0xDA, (byte) 0xA2,
(byte) 0x21, (byte) 0x68, (byte) 0xC2, (byte) 0x34,
(byte) 0xC4, (byte) 0xC6, (byte) 0x62, (byte) 0x8B,
(byte) 0x80, (byte) 0xDC, (byte) 0x1C, (byte) 0xD1,
(byte) 0x29, (byte) 0x02, (byte) 0x4E, (byte) 0x08
};
```

Here is the process in our system with SHA-1 as the hash function to generate the Group Key:

```
/**
To implement a much better churn, generate some random values by getting System time
and adding a little randomness by XORing a few bits of the amount of free memory.
*/
private synchronized byte[] getChurn()
{
  long msecs = System.currentTimeMillis(); // 64 bits
  long fmem = rtime.freeMemory();
```

```
  msecs ^= fmem & 0xFF;

  byte[] tmp = new byte[timeLength];
  // copy into byte array
  for (int i = 0; i < timeLength; i++) {
    tmp[i] = (byte)((msecs >>> (8*(7 - i))) & 0xFF);
  }
  return tmp;
}
```

After calling getChurn() enough to build up sufficient randomness in Randpool, we can now generate random bits from it with the following generator.

```
/**
 * We use SHA-1 to generate requested bytes of random data
 */

public void generateRand(byte[] buffer, int offset, int length)
{
 /**
 * seed is our random number and generates our next random
 * number. Make sure that it  cannot be guessed.
 */

  byte[] churn = getChurn();
  SHA1.update(seed, 0, seedLength);//Hash the seed;
  SHA1.doFinal(churn, seed);
  }
```

The hash function is crucial here. It provides an easy way to generate an arbitrary amount of pseudo-random data without calling getChurn() each time. In effect, the system degrades gracefully from perfect to practical randomness. In this system, it becomes possible to use the result from one generateRand() call to generate a secure Group Key.

This Group Key will be used in the symmetric encryption/decryption: RC4, RC4 is a variable-key-length cipher, with a key that can be anywhere between 8 and 2048 bits long, usually in the range of 48 bits to 128 bits. Here we use RC4 with 128 bits key length.

4.5 Distribute the Group Key to Other Members

The other group members must get the group key before the secure group communication is fully operational. The purposes of other group member initialization are as follows:

- Establish a secure mechanism for the transmission of the group key between the GKM and group members.
- Send the key message and group rekey interval to the other member.

Establishing a secure communication to transmit the group key consists of four messages between the GKM and the other members. Figure 10 presents the handshake procedure.

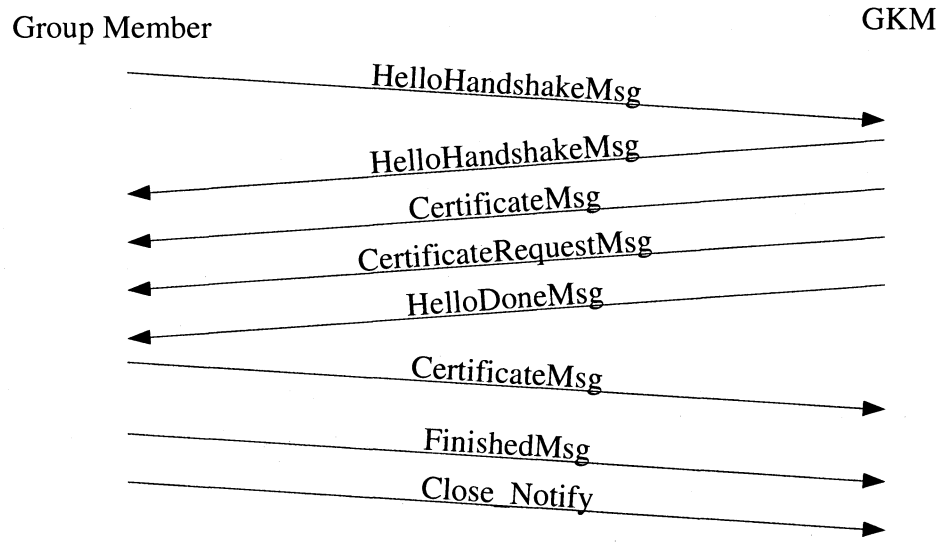

Figure 10. Secure Communication Handshake Messages

1. The initial messages are for the establishment of security pipe between group member and the GKM. The group member sends a *HelloHandshakeMsg* (As shown in figure 11) to accomplish this.

Source ID: Group Member
Destination ID: GKM
Message Type: Hello Handshake

Figure 11. *HelloHandshakeMsg* Format

2. The GKM corresponds to a series of handshake messages. The first message the GKM sends is the *HelloHandshakeMsg*. Next sends its certificate in the *CertificateMsg* (Figure 12). Then it sends a *CertificateRequestMsg* (As shown in figure 13) to the group member. Finally, it sends a *HelloDoneMsg*, which indicates that this phase of the handshake is done. The reason that *HelloDoneMsg* is needed is that some of the more complicated handshake group member receives the *HelloDoneMsg*, it knows that no such other messages will be arriving and so it can proceed with its part of the handshake.

Source ID: GKM/Group Member
Destination ID: Group Member/GKM
Message Type: Certificate
Content: Certificate Content

Figure 12. *CertificateMsg* Format

Source ID: GKM
Destination ID: Group Member
Message Type: Certificate Request

Figure 13. *CertificateRequestMsg* Format

3. Group member corresponds to GKM with *CertificateMsg,* which is initiated by GKM sending the *CertificateRequestMsg* to the group member.

All we have managed to do so far is to authenticate both the GKM and the group member and share some keying material. The purpose of the handshake is to set up the shared state required to make sending and receiving protected data possible.

The GKM must wait until it receives the group member finished message before it can send its first byte of data. It's safest to simply wait for the group member's *FinishedMsg*.

Finally, the group member shuts down the connection, but first it sends a close-notify alert to indicate that the connection is about to close.

After the handshake is finished, the GKM creates an *EncryptedGrpKeyMsg* (As shown in figure 14) which contains the group key generated by the GKM and encrypted with RSA asymmetric method.

Here we use value *GroupKeyVersion* to assure that at the same time, all the group members use the same group key to encrypt/decrypt secure message. When the first Group Key is generated, the value of *GroupKeyVersion* is initialized to 0. When every periodic rekey occurs, this number plus 1.

Source ID: GKM
Destination ID: Group Member
Message Type: Group Key
Message Content: Encrypted Group Key
Due Time
Group Key Version: 0

Figure 14. *EncryptedGrpKeyMsg* Format

Asymmetric encryption relies on two different keys (public key and private key) for secure interactions. GKM has its own private key and the group member's public key; group member has its own private key and the GKM's public key. Figure 15 shows how the group key encryption works.

First, GKM encrypts the group key with the group member's public key and then GKM's private key. Because the encryption is reversible, that is the GKM's private key is used to encrypt, and the GKM's public key, which is kept by the group member, to decrypt. The message is created that essentially prove it came from the owner of the key. The group key is

encrypted with the group member's public key, which is kept by the GKM, can only be decrypted by the group member's private key.

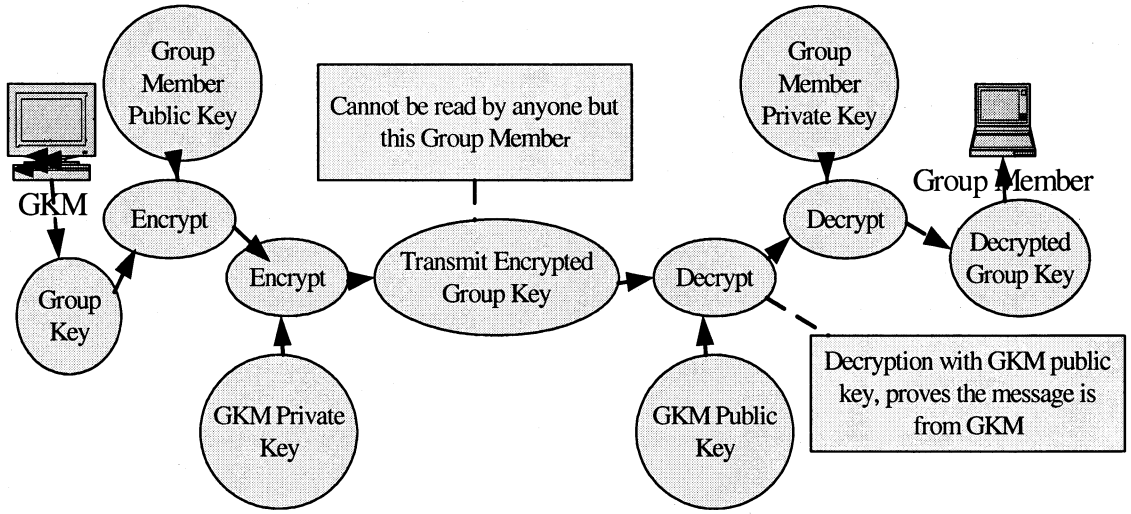

Figure 15. Group Key Encryption/ Decryption Procedure

The JXTA platform supports RSA with PKCS#1 padding in this system. Below illustrates how RSA is implemented in JXTA's crypto suite.

```
RSA rsaAlgorithm = RSA();
SecretKey key1 = GroupMember_Publickey;
SecretKey Key2 = GKM_Privatekey;
byte[] message1 = rsaAlgorithm(byte[] GroupKey, int offset, int length,
                    key1, Boolean encrypt);
byte[] encrypted_GroupKey = rsaAlgorithm(byte[] messag1, int offset,
                                int length, key2, Boolean encrypt);
```

This approach solves the main problem with symmetric key encryption caused by the exchange of what is here a private key because the private keys are never exchanged. Asymmetric encryption consumes more processing time and memory for calculations than symmetric key encryption. The time to run the asymmetric is livable for group key transmission because the group key is not large data.

4.6 Group Rekey

Cryptographic key has a life span. New key must replace "old" group key prior to the end of its cryptographic life. This process is rekey.

Rekey has the advantage of using an existing cryptographic association to distribute key. Also, there is no requirement to verify the identity and authorization for the other members. Identity and authorization are assumed.

A group rekey consists of two stages. First the Group Key Manager creates new group key. Second the new group key is sent to the group members in a broadcast message.

Source ID: GKM
Message Type: Rekey
Content: Encrypted New Group Key
Due Time
Group Key Version

Figure 16. *RekeyMsg* Format

The GKM creates the new group key in exactly the same manner as used during Group Key Generation. GKM creates a *RekeyMsg* (Figure 16). This *RekeyMsg* uses the current group key to encrypt the new key. The encrypted new group key also contains the group key's due time and group key version, which identifies the current group key version and makes all group members to have the same group key at the same time. All members of the group use broadcast to accept this message.

The GKM encrypts the new group key information with the existing group key. Since all group members possess the former group key, the entire group can correctly decrypt this message.

4.7 Send and Receive Secure Messages within the Group

Symmetric encryption is a simple but powerful way of secure communication; it has its disadvantages because of that single way. However, it is a good and practical method if a group of peers needs to send messages that can be read by anyone with a single group key. In figure 17, we can see a message being passed among a group. They are using a single key to encrypt and decrypt a message. Note that the key was exchanged over the group via our group key management protocol. This is very practical for secure group chat where a number of group members communicate if all of the group members have unique key so that a attacker could not sniff secret messages. Symmetric encryption is widely used because it is very fast compared to asymmetric encryption methods. So it becomes increasingly important as the size of the data to be encrypted increases. This project uses a combination of symmetric and asymmetric methods. Asymmetric method is used to securely exchange the group key. After this is done, symmetric encryption is used to encrypt all subsequent exchanges, so we can benefit from the best of worlds.

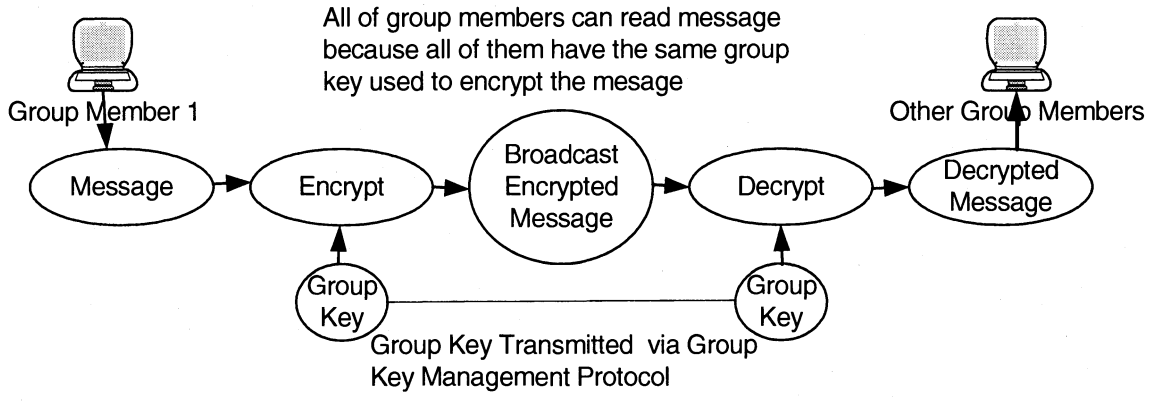

Figure 17. Encrypt/Decrypt message

Figure 18 summarizes the symmetric algorithms, which are common. As we can see, RC4 is by far the fastest algorithm. [6]

Cipher	Key Length	Speed(MB/s)
DES-CBC	56	9
3DES-CBC	168	3
RC2-CBC	Variable	0.9
RC4	Variable	45

Figure 18. Some common cryptographic algorithms (Pentium II 400)

JXTA defines a parent interface for symmetric cryptographic algorithm, RC4. RC4 is a stream cipher and has been received widespread attention. It is extremely fast; a Pentium II/400 can achieve speeds on the order of 45MB/s. Below illustrates how symmetric encryption is implemented in JXTA.

```
JxtaCrypto suite = new JxtaCryptoSuite(JxtaCrypto.MEMBER_RC4,
                        null, (byte)0, (byte)0);
Cipher rc4 = suite.getJxtaCipher();

rc4.init(GroupKey, Cipher.MODE_ENCRYPT);
rc4.doFinal(input.message(), 0, input.length(), byte[] encryptedMessage, 0);
```

The *init()* method initializes the instance of cipher with the group key. When the message is ready, the *doFinal()* method is called. The *doFinal()* method performs the encryption or decryption.

When a group member within the group is ready to broadcast the content, it needs to call encryption method to create a *SecureMsg* as follows:

```
SecureMsg.type = SECURE_GROUP_CHAT;
SecureMsg.byte = encryptedMessage;
SecureMsg.GKversion = GroupKeyVersion;
```

After the group members receive the *SecureMsg*, check the message type and the group key version field. If the version number is equal to its current group key version, it then calls decryption method to decrypt the message content by RC4 with the current group key. If the version number is higher than the current one, it means that this member does not receive the GKM's *RekeyMsg*. Then this member sends a *RekeyRequestMsg* to the GKM, once the GKM gets this message, it will create a *RekeyMsg* and distribut it to this member.

5. EVALUATION

JetMeeting networks are highly complex because they are unlimited in size. In addition, peers are dynamic and unpredictable, as they join and leave frequently. This complexity is difficult to mirror without enormous effort. Instead, we decided to use 12 peers from 3 ISPs, which include Cox, AOL, and Verizon in North Virginia, to evaluate the system. Although this does not reflect the *JetMeeting* complexity, it reveals the general characters of the system.

The system has been continuously tested during its development process. To evaluate its performance, the following test environment including computers and their physical connections are shown in figure 19.

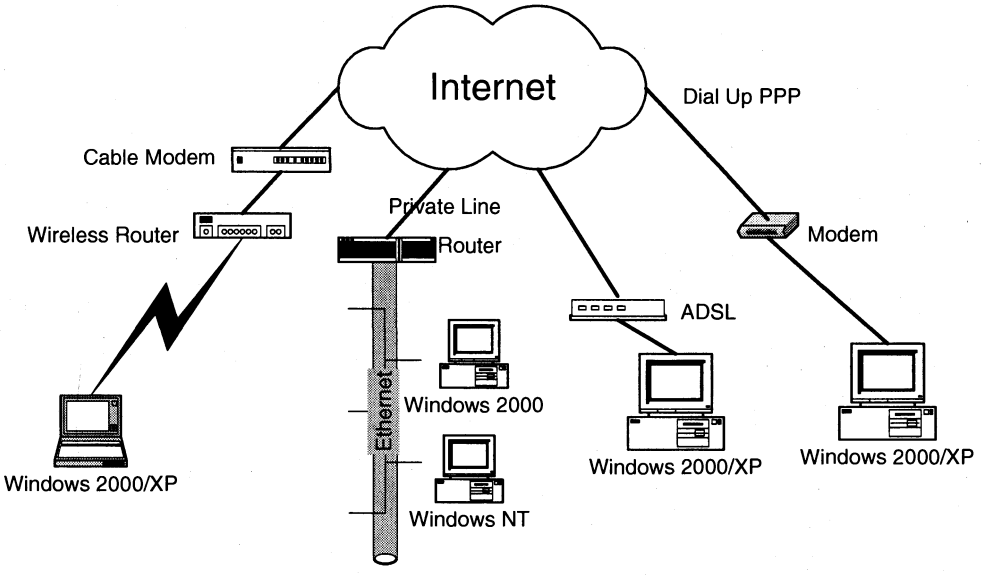

Figure 19. Test Environment

Only one peer runs in a computer. Their OS and other settings are as the followings:

- OS: Windows NT/XP/2000
- CPU: Pentium 4, 1.1G/1.3G/1.8G/2.4G
- JXTA: Stable Builds Version 2.1 released in 6/8/2003
- API: Java JDK1.4.1
- Peer configuration: simple peer using a relay (due to behind firewall)
- Transport protocol: TCP and HTTP (due to behind firewall)

Before forming a group to collaborate each other, all peers have to first connect to the JXTA network. If a peer is properly configured, it will have no problem to connect to the JXTA network. In our testing environment, due to each of peers is behind either a firewall or connected to the Internet via an ISP, every peer needs to set the HTTP protocol with specifying a relay peer and one rendezvous peer on the network.

At the moment, there currently exist two groups: *Cyclone* and *ISU*. Each peer can join either *Cyclone,* which is a secure group, or *ISU,* which is not a secure group.

The system evaluations are discussed from the following aspects:

5.1Transfer Latency

We conducted several experiments to measure the transfer latency using the real public Internet behavior, including latency and bandwidth unpredictability. One of things we measured was the difference in the latency between secure group communication and non-secure group communication to evaluate the speed performance.

We repeated ran the tests at same time of different days to minimize the effect of network congestion. All the tests occurred between 1:00AM and 6:00AM. Below shows the test methods:

(1) Selected $Peer_p$ (p was from 0 to 11) as a Primary Peer, then this peer generated a message, which size was 128 bytes, 256 bytes, 384 bytes, 512 bytes, 768bytes, or 1024 bytes.

(2) $Peer_p$ captured system time to $Time_p$

(3) Encrypted the message with the group key

(4) Broadcasted the encrypted message within the group

(5) Other 11 peers received the encrypted message

(6) Then the peers decrypted it

(7) Replied an empty message to the Primary Peer

(8) When the Primary Peer received the response from $Peer_j$ (j was from 1 to 11), it captured system time to $Time_j$, the difference between $Time_j$ and $Time_p$ was the round trip latency for the encrypted message transferred between Primary Peer and $Peer_j$.

(9) Calculated the average latency $Latency_k = \dfrac{\sum_{j=1}^{11}(Time_p - Time_j)}{11}$

We repeatedly ran the above test every 5 minutes in each peer. After one round, then they started the next round. Totally each peer repeatedly ran 5 times, so all 12 peers totally have 60 *Latency* values for the same message size. The average latency for the same size message was calculated as the following:

$$Average\_Latency = \frac{\sum_{k=1}^{60} Latency_k}{60}$$

To make a better comparison, the first test used encrypted method, which each peer joined *Cyclone* group, which each message was encrypted before it was broadcasted within the group and the message was decrypted before the peer replied a response to the Primary Peer. Next, we tested using the same method, but each peer joined *ISU* group and the message was broadcasted without step 3 and 6. Table 1 shows the test average values in milliseconds (ms).

Message Size	128 Bytes		256 bytes		512 Bytes	
	Encrypted	Non-Encrypted	Encrypted	Non-Encrypted	Encrypted	Non-Encrypted
Average Latency	304	301	304	305	308	309
Message Size	768 Bytes		1024 bytes			
	Encrypted	Non-Encrypted	Encrypted	Non-Encrypted		
Average Latency	332	332	331	333		

Table 1. Average latency of round trip message transfer

The test result shows that all messages have been correctly passed among group members; it also shows that encrypted message transfer and non-encrypted message transfer are both in the same latency. As we expected, the encrypted message transfer speed performance did not degrade, thus demonstrating that the symmetric RC4 encryption method is extremely fast.

The network consumed most message transfer time, which was not consumed by the encrypting/decrypting process.

5.2 Fault Tolerance

Fault tolerance is a system's ability to supply regular service operations in the presence of hardware or software faults. In principle, the absence of centralized control and coordination makes P2P systems robust with respect to failures that might occur at any peer. Faults at client peers don't usually affect system behavior, but faults at server peers can result in data loss.

As part of the test, we simulated random failures in these 12 peers except to the Group Key Manager peer. When one peer failed, as expected, the secure group communication worked normally.

As we simulated that the GKM failed to work, if no new peer joined the group, the secure group communication still could work with the existing Group Key. But if a new peer joined the group after the GKM was down, because it would send an inquiry message to check who was the GKM, at this moment, no one would answer the message, so the newly joined peer would elected itself as the GKM and generated a new GK. The new group member would encrypt the messages with the new GK; eventually other group members could not correctly decrypt the message with their GK. The same situation would happen when the new group member received an encrypted message from other members. These faults in GKM can result in data loss.

5.3 Scalability

Scalability is the degree of adaptability that a system exhibits with respect to increasing-load situations. JetMeeting is dynamic and unpredictable in size (growing or shrinking), topology (connecting or disconnecting peers), and activity (number or kind of requests). To ensure high quality of service, issues such as adaptability with respect to the number of peers, must be addressed.

In these experiments, we selected a computer, which equipped with a 1.7GHZ Intel Pentium 4 1.8GHz Processor, 512Mbytes RAM and a 10Mbps cable Internet Access, to be the Group Key Manager. During the experiments, the GK distribution time was observed depending on the number of joining group members.

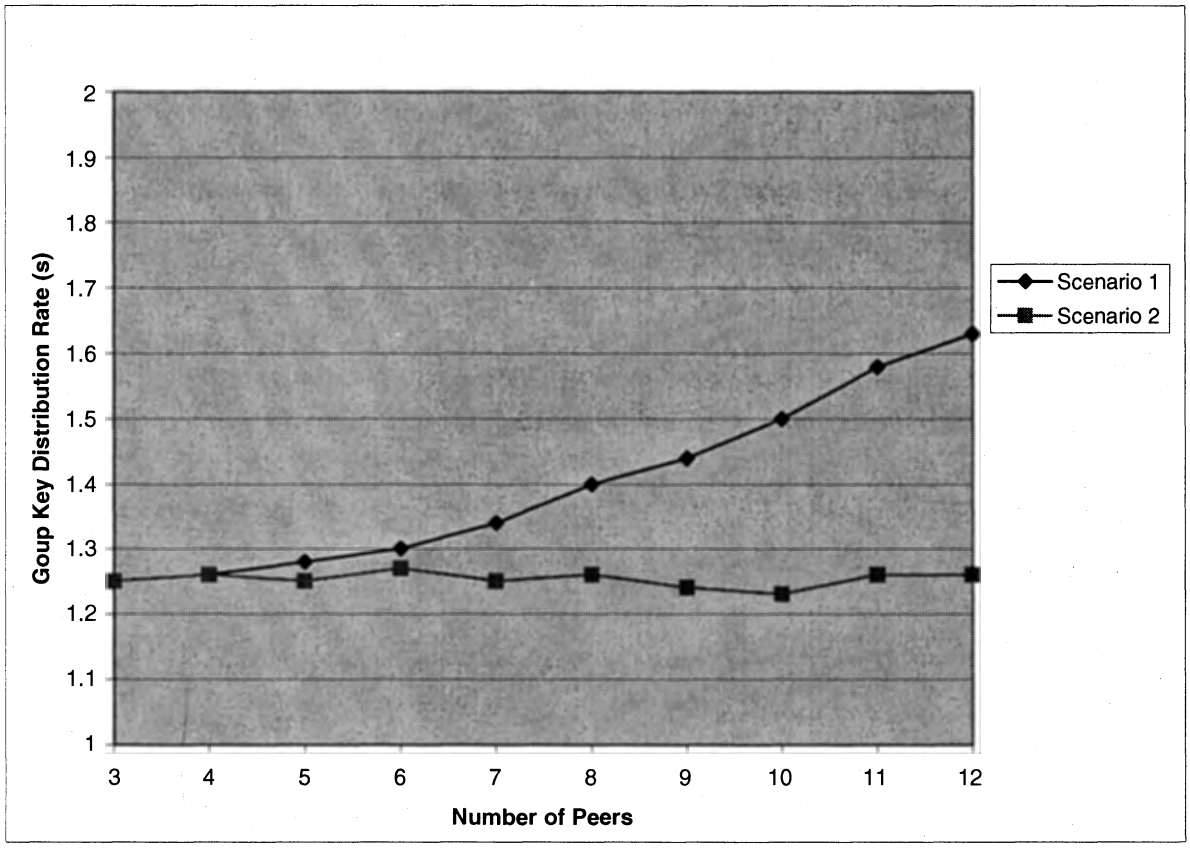

Figure 20. Scalability: group key distribution rate growth/reduction with increasing number of peers.

Figure 20 illustrates the scalability behavior of the system during the experiments. Based on the results of testing with three joining group members, it was measured whether the group key distribution time, which was defined the average time difference between the WhoIsGKMMsg sent and the EncryptedGrpKeyMsg received by the group member, increased or decreased when further peers joined the group communication.

Scenario 1, if all the other 9 peers joined the group within 3 minutes, the group key distribution time increased with an increasing number of peers, which is a sign for limited

scalability. This can be explained by the group key distribution model, which is managed by the centralized GKM. Rushed traffic cause message collisions because GK handling workload is taken only by one GKM.

Scenario 2, if all the peers randomly joined the group, the group key distribution time did not increase with an increasing number of peers, which demonstrates that GK distribution is scalable if not all the peers join the group within a short time. The centralized GKM is not the bottleneck even though the number of group member grows big but not fast.

6. CONCLUSION: ACCOMPLISHMENT, LIMITATIONS AND FUTURE WORK

In this thesis, we presented a system for secure multicasting, which is developments that focus on specific aspects in a dynamic P2P application with aiming to deliver real systems. The core of the system consists of four functions: Group Key Manager election, Group Key generation, Group Key distribution and secure communications. The thesis also evaluates the system's efficiency, scalability, and fault tolerance. The results of the evaluation supported our claims regarding efficiency and scalability.

In the core of security, our system is based on mechanisms for protecting sensitive communications and resources of a group member via encryption and authentication techniques. This Group Key Management protocol is tested to be efficient and scalable and can be used to highly dynamic and unpredicted group sizes group chat and collaboration applications based on P2P.

However, some limitations apply to our system. The Group Key Manager is centralized, and the GKM election is simple. Malicious users can exploit this mechanism. Another issue is that of the group key generation, which is not changed when a peer joins or leaves the group. Former malicious group members can be attackers to the secure multicast communications. In addition, when GKM starts to operate incorrectly or is down, the faults at GKM can result in data loss.

To make JetMeeting a reliable and secure application, some considerations deserve further developments:

- Reliable handling the GKM's abnormal leave from a group
- Looking for an efficient approach to generate a Group Key and distribute it to only the members of the group affected by a change
- Further enhancing the performance of the group key distribution

7. REFERENCES

[1] SUN Microsystems, Project JXTA, http://www.sun.com/jxta/, 2002.

[2] SUN Microsystems, JXTA programmer's guide, http://www.jxta.org/docs/jxtaprogguide_final.pdf, 2003.

[3] J. D. Gradecki, "Mastering JXTA: Building Java Peer-to-Peer Applications," Wiley Publishing,Inc, 2002.

[4] D. Brookshier, D. Govoni, N. Krishnan, and J. Soto, "JXTA: Java P2P Programming," Sams, 2002.

[5] W. Jim, S. Graupner, and A. Sahai, "A Secure Platform for Peer-to-Peer Computing in the Internet," Proceedings of the 35th Hawaii International Conference on System Science, 2002.

[6] E. Rescorla, "SSL and TLS: Designing and Building Secure System," Addison-Wesley, 2000.

[7] R. Atkinson, "Security Architecture for the Internet Protocol," RFC 1825, August 1995.

[8] H. Harney, and C. Muckenhirn, "Group Key Management Protocol (GKMP) Specification," RFC 2093, July 1997.

[9] H. Harney, and C. Muckenhirn, "Group Key Management Protocol (GKMP) Architecture," RFC 2094, July 1997.

[10] D. Harkins and D. Carrel, "The Internet Key Exchange," RFC 2409, 1998.

[11] D. Eastlake, S. Crocker, and J. Schiller, "Random Recommendation for Security," RFC 1750, December 1994.

[12] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile," RFC 2459, January 1999.

[13] H. Orman, "The OAKLEY Key Determination Protocol," RFC 2412, November 1998.

[14] B. Schneier, "Applied Cryptography: Protocols, Algorithms, and Source Code in C," 2nd Edition, New York Wiley, 1996